A dependently-typed formalization of $\lambda^{\rightarrow}$, substitution, denotation, normalization.

Draft of June 18, 2007

Matthieu Sozeau
Univ. Paris Sud, CNRS, Laboratoire LRI, UMR 8623, Orsay, F-91405
INRIA Futurs, ProVal, Parc Orsay Universit, F-91893
sozeau@lri.fr

We will develop a complete formalization of simply-typed lambda-calculus with constants in Coq, using the Program extension to write dependently-typed programs on inductive families. The development uses a pure de Bruijn encoding and dependently-typed abstract syntax. It includes the theory of lifting and substitution, a denotational semantics given by interpreting terms directly into Coq, and finally a proof of weak normalization for the call-by name strategy.

This paper describes all the technical parts of the development. It is a literate coqdoc script, missing only the proof scripts.

First we define a set of tactics that will be used later.

Rewrite using uniqueness of indentity proofs $H = \mathit{refl\_equal}\ X$.

```
Ltac simpl_uip :=
  match goal with
    [ H : ?X = ?X ⊢ _ ] ⇒ rewrite (UIP_refl _ _ H) in ×; clear H
  end.
```

Try to abstract a proof of equality, if no proof of the same equality is present in the context.

```
Ltac abstract_eq_hyp H' p :=
  match type of p with
    ?X = ?Y ⇒
    match goal with
      | [ H : X = Y ⊢ _ ] ⇒ fail 1
      | _ ⇒ set (H':=p) ; clearbody H'
    end
  end.
```

Apply the tactic tac to proofs of equality appearing as coercion arguments.

```
Ltac on_coerce_proof tac :=
```

```
  match goal with
    [ | - ?T ] ⇒
    match T with
      | context [ eq_rect ____?p ] ⇒ tac p
    end
  end.
```

Abstract proofs of equalities of coercions.

```
Ltac abstract_eq_proof := on_coerce_proof ltac:(fun p ⇒ let H := fresh "eqH"
in abstract_eq_hyp H p).
```

Factorize proofs, by using proof irrelevance so that two proofs of the same equality in the goal become convertible.

```
Ltac pi_eq_proof_hyp p :=
  match type of p with
    ?X = ?Y ⇒
    match goal with
      | [ H : X = Y ⊢ _ ] ⇒
        match p with
          | H ⇒ fail 2
          | _ ⇒ rewrite (proof_irrelevance (X = Y) p H)
        end
      | _ ⇒ fail " No hypothesis with same type "
    end
  end.
```

Factorize proofs of equality appearing as coercion arguments.

```
Ltac pi_eq_proof := on_coerce_proof pi_eq_proof.
```

Clear unused reflexivity proofs.

```
Ltac clear_refl_eq :=
  match goal with [ H : ?X = ?X ⊢ _ ] ⇒ clear H end.
Ltac clear_refl_eqs := repeat clear_refl_eq.
```

Clear unused equality proofs.

```
Ltac clear_eq :=
  match goal with [ H : _ = _ ⊢ _ ] ⇒ clear H end.
Ltac clear_eqs := repeat clear_eq.
```

Everything is done in implicit arguments mode.

1 Constants

The development is parameterized by module implementing the constants and their typing.

```
Module Type Constants.
```

The set of constant types.

```
  Parameter constant_types : Set.
```

The constants themselves.

```
  Parameter constants : Set.
```

A typing function for constants.

```
  Parameter type_constant : constants → constant_types.

End Constants.
```

2 The implementation.

The functor Λ implements typing, denotational semantics and normalization of simply-typed lambda-calculus plus constants C.

```
Module Λ (C : Constants).
  Import C.
```

2.1 Abstract syntax

The types are just atoms or the arrow.

```
  Inductive type : Set :=
  | type_cst : constant_types → type
  | type_arrow : type → type → type.

  Coercion type_cst : constant_types ¿-¿ type.
  Infix " → " := type_arrow (at level 20).
```

A context is either empty or a context and a variable. Contexts are effectively snoc-lists, hence the new inductive.

```
  Inductive ctx : Set :=
  | [] : ctx
  | comma : ctx → type → ctx.

  Infix "," := comma (at level 40, left associativity).
```

Appending one context to another, by induction on the second one.

```
  Fixpoint append (Γ Δ : ctx) { struct Δ } : ctx :=
    match Δ with
      | [] ⇒ Γ
      | Δ' , x ⇒ (append Γ Δ') , x
    end.
```

```
Hint Unfold append.

Infix ";" := append (right associativity, at level 60).
```

Due to the way append is defined, these two are not convertible.

```
Lemma app_empty : ∀ Γ, ([] ; Γ) = Γ.
```

Associativity of context appending.

```
Lemma app_assoc : ∀ Γ Δ Λ, Γ ; Δ ; Λ = (Γ ; Δ) ; Λ.

Hint Rewrite app_empty app_assoc : app.
Opaque app_assoc.
```

Simplification tactic using the previous lemmas.

```
Ltac my_simpl := subtac_simpl ; try simpl_JMeq ; autorewrite with app ; subtac_simpl.
```

Variables are encoded as typed de Bruijn indices.

```
Inductive var : ctx → type → Set :=
| first : ∀ Γ T, var (Γ , T) T
| next : ∀ Γ T, var Γ T → ∀ U, var (Γ , U) T.
```

The AST of well-typed terms. Constants get their type from the *type_constant* function. A rel just embeds a variable. The abstraction constructor lam contains a body typed in an extended environment. Application applies only functions to objects having their domain type.

```
Inductive term (Γ : ctx) : type → Set :=
| cst : ∀ c : constants, term Γ (type_constant c)
| rel : ∀ T, var Γ T → term Γ T
| lam : ∀ (T T' : type),
  term (Γ , T) T' → term Γ (T → T')
| app : ∀ T T' : type,
  term Γ (T → T') → term Γ T → term Γ T'.
```

The really dependent scheme for term. Not used in the development but may be useful.

```
Scheme term_dep := Induction for term Sort Prop.
```

Tactic to automatically invert contradictory hypotheses.

```
Ltac try_inversion :=
  match goal with
    | H : _ = _ ⊢ _ ⇒ inversion H ; auto
    | H : var [] _ ⊢ _ ⇒ inversion H
  end.
```

Use these for solving obligations from now on.

```
Obligations Tactic := my_simpl ; try try_inversion ; try omega.
```

2.2 Dealing with coercions and equality.

The next two functions reindex variables and terms in Γ to a new context Γ' with respect to an equality between Γ and Γ'. These are dynamic coercions in the sense that they can be reduced when applied to a constructed term, contrary to the substitution principle *eq_rect*.

```
Program Fixpoint coerce_var_context Γ T (v : var Γ T) Γ' (H : Γ = Γ')
  { struct v } : var Γ' T :=
  match Γ' with
    | [] ⇒ !
    | Γ'' , V' ⇒
      match v with
        | first vG vT ⇒ first Γ'' V'
        | next v'G v'T v' V ⇒
          @next Γ'' v'T (coerce_var_context v' _) V'
      end
  end.

Program Fixpoint coerce_term_context Γ T (t : term Γ T) Γ' (H : Γ = Γ')
  { struct t } : term Γ' T :=
  match t with
    | cst c ⇒ cst Γ' c
    | rel T v ⇒ rel (coerce_var_context v H)
    | lam τ τ' b ⇒ @lam Γ' _ _ (@coerce_term_context (Γ , τ) τ' b (Γ' , τ) _)
    | app τ τ' f e ⇒ @app Γ' τ τ' (coerce_term_context f H)
      (coerce_term_context e H)
  end.
```

Like *eq_rect*, coerce reduces to the identity when Γ and Γ' are convertible.

```
Lemma coerce_var_context_id : ∀ (Γ : ctx) T (v : var Γ T),
  coerce_var_context v (refl_equal Γ) = v.


Lemma coerce_term_context_id : ∀ (Γ : ctx) T (t : term Γ T),
  coerce_term_context t (refl_equal Γ) = t.
```

We now define some tactics that help to deal with coercions. They work in multiple passes. First, we abstract any equality proof appearing in the goal as an argument of a coercion, so as not to polute it with irrelevant proof terms. Then we factorize the proofs and normalize the term using proof irrelevance so that two references to the same equality proof are the same. Next we try to simplify the equations using Streicher's axiom K (a proof p of $x = x$ can be substituted by *refl_equal* x). Finally we try to rewrite the term using the previous lemmas about *coerce* and *refl_equal*. This procedure is very effective, most of the coercions can be eliminated this way. We get stuck only when an induction is needed.

Abstract a proof of equality, if none is already present in the context.

```
Ltac on_coerce_proof tac :=
  match goal with
    [ | - ?T ] ⇒
    match T with
      | context [ coerce_var_context _?p ] ⇒
        tac p
      | context [ coerce_term_context _?p ] ⇒
        tac p
      | context [ eq_rect ____?p ] ⇒
        tac p
    end
  end.

Ltac abstract_coerce_proof := on_coerce_proof
  ltac:(fun p ⇒
    let H := fresh "coerceH" in
      abstract_eq_hyp H p).

Ltac abstract_coerce_proofs := repeat abstract_coerce_proof.
```

Use proof-irrelevance to eliminate remaining non-abstracted proofs.

```
Ltac pi_coerce_proof := on_coerce_proof ltac:(pi_eq_proof_hyp).

Ltac pi_coerce_proofs := repeat pi_coerce_proof.
```

The two preceding tactics in sequence.

```
Ltac clear_coerce_proofs :=
  abstract_coerce_proofs ; pi_coerce_proofs.
```

Rewrite $coerce\_\times\ t\ (refl\_equal\ \_)$ to t.

```
Hint Rewrite coerce_term_context_id coerce_var_context_id : coerce_id.

Ltac rewrite_coerce_id := autorewrite with coerce_id.
```

Clear the context and term of equality proofs.

```
Ltac clear_coerce_ctx :=
  rewrite_coerce_id ; clear_coerce_proofs.
```

Reapeated elimination of *eq_rect* applications. Abstracting equalities makes it run much faster than an naive implementation.

```
Ltac simpl_eqs :=
  repeat (real_elim_eq_rect ; simpl ; clear_coerce_ctx).
```

Combine all the tactics to simplify goals containing coercions.

```
Ltac simpl_term :=
  simpl ; simpl_eqs ; clear_coerce_ctx ; clear_refl_eqs ;
    try subst ; simpl ; repeat simpl_uip ; rewrite_coerce_id.
```

In fact *eq_rect* and *coerce* are always equal, so we can switch between frozen and dynamic coercions at will.

```
Lemma eq_rect_coerce : ∀ Γ T (t : term Γ T) Γ' (Heq: Γ = Γ'),
  eq_rect _ (fun Γ' ⇒ term Γ' T) t _ Heq = coerce_term_context t Heq.

Lemma eq_rect_coerce_var : ∀ Γ T (t : var Γ T) Γ' (Heq: Γ = Γ'),
  eq_rect _ (fun Γ' ⇒ var Γ' T) t _ Heq = coerce_var_context t Heq.
```

Tactic to change *eq_rect* applications to coercions. Allows to reduce coercions sometimes.

```
Hint Rewrite eq_rect_coerce eq_rect_coerce_var : eq_rect_coerce.

Ltac rew_eq_coerce := autorewrite with eq_rect_coerce.
```

3 Lifting and substitution.

We now define lifting and substitution of terms. The specifications of the functions are exactly the lemmas you would expect.

```
Section Substitution.
```

Lifting a variable t in context Γ ; Δ by U to get a variable in Γ , U ; Δ. The context Δ represents the bindings we must avoid to capture.

```
Program Fixpoint lift_var (Γ Δ : ctx) (T : type) (t : var (Γ ; Δ) T)
  (U : type) {struct t} : var (Γ , U ; Δ) T :=
  match Δ with
    | [] ⇒ next t U
    | Δ' , _ ⇒
      match t with
        | first Γ' _ ⇒ first (Γ , U ; Δ') T
        | next Γ' T' v V' ⇒ next (lift_var Γ Δ' v U) V'
      end
  end.
```

We prove the defining equations as lemmas.

```
Lemma lift_var_empty : ∀ Γ T U (v : var Γ T), lift_var Γ [] v U = next v U.
```

Lifting a term by a type U is just a fold.

```
Program Fixpoint lift_rec Γ Δ T (t : term (Γ ; Δ) T) U
  {struct t}: term (Γ , U ; Δ) T :=
  match t with
    | cst c ⇒ cst _ c
    | rel T v ⇒ rel (lift_var Γ Δ v U)
    | lam τ τ' b ⇒ lam (lift_rec Γ (Δ , τ) b U)
    | app τ τ' f e ⇒ app (lift_rec Γ Δ f U) (lift_rec Γ Δ e U)
  end.
```

The non recursive version, a.k.a. weakening.

```
Program Definition lift (Γ : ctx) T (t : term Γ T) U : term (Γ , U) T :=
  lift_rec Γ [] t U.
```

The defining equations for the variable case.

```
Lemma lift_rec_empty : ∀ Γ T U (v : var Γ T),
  lift_rec Γ [] (rel v) U = rel (next v U).

Lemma lift_rec_comma_first : ∀ Γ T U Δ,
  lift_rec Γ (Δ , T) (rel (first (Γ ; Δ) T)) U = rel (first (Γ , U ; Δ) T).

Lemma lift_rec_comma_next : ∀ Γ T U Δ T' (v : var (Γ ; Δ) T'),
  lift_rec Γ (Δ , T) (rel (next v T)) U = rel (next (lift_var _ _ v U) T).
```

Lifting by a context is lifting by each variable in turn.

```
Program Fixpoint lift_ctx (Γ Δ : ctx) T (t : term Γ T)
  { struct Δ } : term (Γ ; Δ) T :=
  match Δ with
    | [] ⇒ t
    | Δ' , U ⇒ let t' := lift_ctx Δ' t in
      lift t' U
  end.
```

Substitution of a variable by a term in context Γ, under a context Δ.

```
Program Fixpoint subst_var_rec Γ Δ T U (t : var (Γ , T ; Δ) U)
  (s : term Γ T) { struct Δ } : term (Γ ; Δ) U :=
  match Δ with
    | [] ⇒
      match t with
        | first tG tT ⇒ s (** substitution takes place *)
        | next tG' tT' tv _ ⇒
          rel tv (** unlift the var to account for subtitution *)
      end
    | Δ' , V ⇒
      match t with
        | first tG tT ⇒ rel (first (Γ ; Δ') V) (** do nothing *)
        | next tG' tT' tv V ⇒ (** substitute inside, then lift by one *)
          lift (@subst_var_rec Γ Δ' T U tv s) V
      end
  end.
```

The defining equations.

```
Lemma subst_var_rec_empty_first : ∀ Γ T (s : term Γ T),
  subst_var_rec [] (first Γ T) s = s.

Lemma subst_var_rec_empty_next : ∀ Γ T (v : var Γ T) T' (s : term Γ T'),
  subst_var_rec [] (next v T') s = rel v.
```

```
Lemma subst_var_rec_comma_first : ∀ Γ T T' (t : term Γ T') Δ,
  subst_var_rec (Δ , T) (first (Γ , T' ; Δ) T) t = rel (first (Γ ; Δ) T).

Lemma subst_var_rec_comma_next : ∀ Γ T T' (t : term Γ T') Δ
  T" (v : var (Γ , T' ; Δ) T),
  subst_var_rec (Δ , T") (next v T") t = lift (subst_var_rec Δ v t) T".
```

Non-recursive wrapper to substitute the first variable in the context.

```
Program Definition subst_var Γ T U (t : var (Γ , T) U) (s : term Γ T) :
  term Γ U := subst_var_rec [] t s.
```

Substitution in a term.

```
Program Fixpoint subst_rec Γ Δ T U (t : term (Γ , T ; Δ) U)
  (s : term Γ T) { struct t } : term (Γ ; Δ) U :=
  match t with
    | cst c ⇒ cst (Γ ; Δ) c
    | rel _ v ⇒ @subst_var_rec Γ Δ T U v s
    | lam τ τ' b ⇒
      lam (subst_rec (Δ , τ) b s)
    | app τ τ' f e ⇒ app (subst_rec Δ f s) (subst_rec Δ e s)
  end.
```

Wrapped again.

```
Program Definition subst (Γ : ctx) (T U : type) (t : term (Γ , T) U)
  (s : term Γ T) : term Γ U := subst_rec [] t s.
```

The defining equations.

```
Lemma subst_rec_empty_first : ∀ Γ T (t : term Γ T),
  subst_rec [] (rel (first Γ T)) t = t.

Lemma subst_rec_empty_next : ∀ Γ T T' (t : term Γ T) (v : var Γ T'),
  subst_rec [] (rel (next v T)) t = rel v.

Lemma subst_rec_comma_first : ∀ Γ T T' (t : term Γ T') Δ,
  subst_rec (Δ , T) (rel (first (Γ , T' ; Δ) T)) t = rel (first (Γ ; Δ) T).

Lemma subst_rec_comma_next : ∀ Γ T T' (t : term Γ T') Δ
  T" (v : var (Γ , T' ; Δ) T),
  subst_rec (Δ , T") (rel (next v T")) t = lift (subst_rec Δ (rel v) t) T".
```

3.1 Commutation lemmas.

We now relate lifting and substitution to ultimately prove the substitution lemma.

Substituting for a variable just lifted is a no-op.

We first typecheck the statement, which will insert coercions around t to get the right indices. The obligations are automatically resolved using the *Heq* hypothesis.

Note that we separate the equality so that induction can proceed on t of type $\mathsf{term}\ \Gamma'\ T$ while still retaining the equation between Γ' and $\Gamma\ ;\ \Delta$.

```
Program Definition subst_lift_rec_stmt :=
  ∀ Γ' T (t : term Γ' T) Γ Δ T' (u : term Γ T'), (Heq : Γ' = (Γ ; Δ)),
    subst_rec Δ (lift_rec Γ Δ t T') u = t.
```

Now we prove it, by induction on t. We first rewrite *eq_rect* to coercions, then simplify the term using *simpl_term*. This takes care of almost all the plumbing.

```
Lemma subst_lift_rec : subst_lift_rec_stmt.
```

The real lemma is then just an instance of the previous one.

```
Lemma subst_lift : ∀ Γ Δ T (t : term (Γ ; Δ) T) T' (u : term Γ T'),
  subst_rec Δ (lift_rec Γ Δ t T') u = t.
```

We now prove that equality coercion commute with lift to solve some of the later lemmas which involve coercions of lift calls. Idealy this should be either easily derivable or automatically generated by some kind of generic programming.

```
Lemma coerce_var_context_lift_var :
  ∀ Γ'' T (t : var Γ'' T) Γ Δ (Heq:Γ'' = Γ ; Δ)
    Γ' U (H : (Γ , U ; Δ) = (Γ' , U ; Δ)) (H' : Γ'' = (Γ' ; Δ)),
    coerce_var_context (lift_var Γ Δ (coerce_var_context t Heq) U) H =
    lift_var Γ' Δ (coerce_var_context t H') U.

Lemma coerce_term_context_lift_rec_aux :
  ∀ Γ'' T (t : term Γ'' T) Γ Δ (Heq:Γ'' = Γ ; Δ) Γ' U
    (H : (Γ , U ; Δ) = (Γ' , U ; Δ)) (H' : Γ'' = (Γ' ; Δ)),
    coerce_term_context (lift_rec Γ Δ (coerce_term_context t Heq) U) H =
    lift_rec Γ' Δ (coerce_term_context t H') U.

Lemma coerce_term_context_lift_rec :
  ∀ Γ Δ T (t : term (Γ ; Δ) T) Γ' U (H : (Γ , U ; Δ) = (Γ' , U ; Δ))
    (H' : (Γ ; Δ) = (Γ' ; Δ)),
    coerce_term_context (lift_rec Γ Δ t U) H =
    lift_rec Γ' Δ (coerce_term_context t H') U.
```

We have to take care of the opaqueness of these constants all the time, spurious unfoldings can occur otherwise, greatly obfuscating the goal.

```
Opaque coerce_var_context coerce_term_context.
Opaque lift_rec subst_rec.
```

We continue on our road to the substitution lemma. Here we prove commutation of *lift_var*.

```
Program Definition lift_lift_var_stmt :=
  ∀ Γ' U (t : var Γ' U) Γ Δ Δ' (Heq : Γ' = Γ ; Δ ; Δ') T T',
    lift_var (Γ , T ; Δ) Δ' (lift_var Γ (Δ ; Δ') t T) T' =
    lift_var Γ (Δ , T' ; Δ') (lift_var (Γ ; Δ) Δ' t T') T.
```

```
Lemma lift_lift_var : lift_lift_var_stmt.

Program Definition lift_lift_stmt :=
  ∀ Γ' U (t : term Γ' U) Γ Δ Δ' T T' (Heq: Γ' = Γ ; Δ ; Δ'),
  lift_rec (Γ , T ; Δ) Δ' (lift_rec Γ (Δ ; Δ') t T) T' =
  lift_rec Γ (Δ , T' ; Δ') (lift_rec (Γ ; Δ) Δ' t T') T.

Lemma lift_lift : lift_lift_stmt.
```

Commutation of lifting and substitution, for variables.

```
Program Definition subst_lift_var_comm_stmt :=
  ∀ Γ' T (t : var Γ' T) Γ Δ' Δ U T' (s : term Γ U)
    (Heq: Γ' = Γ , U ; Δ ; Δ'),
    subst_var_rec (Δ , T' ; Δ') (lift_var (Γ , U ; Δ) Δ' t T') s =
    lift_rec (Γ ; Δ) Δ' (subst_var_rec (Δ ; Δ') t s) T'.
```

Commutation of lifting and substitution, for terms.

```
Program Definition subst_lift_comm_stmt :=
  ∀ Γ' T (t : term Γ' T) Γ U Δ Δ' T' (s : term Γ U)
    (Heq: Γ' = Γ , U ; Δ ; Δ'),
    subst_rec (Δ , T' ; Δ') (lift_rec (Γ , U ; Δ) Δ' t T') s =
    lift_rec (Γ ; Δ) Δ' (subst_rec (Δ ; Δ') t s) T'.
```

The non-recursive case, with no Δ' context. This time the statement need not contain any coercions.

```
Lemma subst_lift_comm_full :
  ∀ Γ U Δ T (t : term (Γ , U ; Δ) T) (s : term Γ U) T',
    subst_rec (Δ , T') (lift_rec (Γ , U ; Δ) [] t T') s =
    lift_rec (Γ ; Δ) [] (subst_rec Δ t s) T'.
```

Finally, we can state the subtitution lemma as usual. The proof uses all the lemmas proven above, including commutation of *lift_rec* and *coerce_term_context*.

```
Program Definition subst_rec_comm_stmt :=
  ∀ Γ' T (u : term Γ' T) Δ' Γ Δ T' T''
    (s : term (Γ , T'' ; Δ) T') (t : term Γ T'')
    (Heq: Γ' = (Γ , T'' ; Δ , T' ; Δ')),
    subst_rec Δ' (subst_rec (Δ , T' ; Δ') u t) (subst_rec Δ s t) =
    subst_rec (Δ ; Δ') (subst_rec Δ' u s) t.

Lemma subst_rec_comm : subst_rec_comm_stmt.
```

Non-recursive case, where no coercion is needed anymore.

```
Lemma subst_comm :
  ∀ Γ T'' Δ T' T (u : term (Γ , T'' ; Δ , T') T)
    (s : term (Γ , T'' ; Δ) T') (t : term Γ T''),
    subst (subst_rec (Δ , T') u t) (subst_rec Δ s t) =
```

```
    subst_rec Δ (subst u s) t.

End Substitution.
```

4 Interpretation into CIC.

There is a direct embedding of $\lambda^{\rightarrow}$ into the Calculus of Constructions, which we build now.

```
Section Interpretation.
```

We require interpretations for the constants and their types.

```
Variable interp_cst_type : constant_types → Set.
Variable interp_cst : ∀ c : constants, interp_cst_type (type_constant c).
```

Interpret a type into a coq `Set`. We need an impredicate `Set` to work with this definition.

```
Fixpoint interp_type (t : type) : Set :=
  match t with
    | type_cst c ⇒ interp_cst_type c
    | a → b ⇒ interp_type a → interp_type b
  end.
```

A valuation is a function which associates a term to each variable in a context.

```
Program Definition valuation (Γ : ctx) :=
  ∀ T, var Γ T → interp_type T.
```

We can extend a valuation by a term.

```
Program Definition augment_valuation (Γ : ctx) (rho : valuation Γ)
  (T : type) (x : interp_type T) : valuation (Γ , T) :=
  fun T v ⇒
    match v with
      | first _ _ ⇒ x
      | next Γ' T' v' _ ⇒ rho T' v'
    end.
```

Now we can interpret terms easily.

```
Fixpoint interp_term (Γ : ctx) (T : type) (t : term Γ T)
  (v : valuation Γ) {struct t} : interp_type T :=
  match t in term _ T return valuation Γ → interp_type T with
    | cst c ⇒ fun _ ⇒ interp_cst c
    | rel T n ⇒ fun v ⇒ v T n
    | lam T T' b ⇒ (fun v ⇒
      (fun x : interp_type T ⇒
        interp_term b (augment_valuation v x)))
    | app T T' f e ⇒ fun v ⇒ (interp_term f v) (interp_term e v)
```

```
  end v.
```

The empty valuation.

```
  Program Definition nil_valuation : valuation [] := fun n v ⇒ !.
```

Interpretation of closed terms.

```
  Definition interp_closed_term (T : type) (t : term [] T) : interp_type T :=
    interp_term t nil_valuation.

End Interpretation.
```

5 A proof of weak normalization.

Inspired by (Biernacka *et al.*, 2006), we build a proof of weak-head normalization using a tait-style proof. This is an instance of normalization by evaluation where the semantic domain is given by R, the strong computability predicate.

```
Section Normalization.
```

Beta-reduction and its contextual closure.

```
Inductive reduces (Γ : ctx) : ∀ T : type, term Γ T → term Γ T → Prop :=
| red_beta : ∀ τ τ' (b : term (Γ , τ) τ') (e : term Γ τ),
    reduces (app (lam b) e) (subst b e)
| red_app : ∀ τ τ' f f', @reduces Γ (τ → τ') f f' →
    ∀ e, reduces (app f e) (app f' e).
```

Evaluation gives the definition of normal forms, either:

- *eval_trans* If a term t reduces to s which itself evaluates to r then t evaluates to r. So evaluation is "backward-closed" by reduction.
- *eval_lam* Lambda expressions are all in normal form.
- *eval_cst* A constant is a normal form.

```
Inductive evaluates (Γ : ctx) : ∀ T : type, term Γ T → term Γ T → Prop :=
| eval_trans : ∀ T (t : term Γ T) s r,
    reduces t s → evaluates s r → evaluates t r
| eval_lam : ∀ τ τ' b, @evaluates Γ (τ → τ') (lam b) (lam b)
| eval_cst : ∀ c, evaluates (cst Γ c) (cst Γ c).
```

R is the glueing model in the sense of Coquand et al. We have a syntactic part v which is the value of the term t and a semantic one given by the function in the arrow case which ensures closure by application. It is trivial for constants. Note that R lives in `Set` so that we can extract it later.

```
Program Fixpoint R Γ T (t : term Γ T) { struct T } : Set :=
  match T with
    | type_cst c ⇒ { v : term Γ T | evaluates t v }
```

```
    | τ → τ' ⇒
      ({v : term Γ T | evaluates t v } ×
        (∀ s, R s → R (@app Γ τ τ' t s)))%type
  end.
```

Extract the "syntactic" part.

```
Lemma R_eval : ∀ Γ T (t : term Γ T), R t → { v : term Γ T | evaluates t v }.
```

Backward-closedness of evaluation extends to computability.

```
Lemma red_R_R : ∀ Γ T (t s : term Γ T), reduces t s → R s → R t.
```

5.1 Telescopes.

A substitution is a telescope of terms.

```
Inductive substitution : ctx → Set :=
| SNil : substitution []
| SCons : ∀ Γ, substitution Γ → ∀ T, term Γ T → substitution (Γ , T).
```

Substitute a term by a telescope.

```
Program Fixpoint mult_subst Γ Δ U (u : term (Γ ; Δ) U)
  (s : substitution Γ) { struct s } : term Δ U :=
  match s with
    | SNil ⇒ u
    | SCons Γ' s' T t ⇒
      mult_subst Δ (subst_rec Δ u t) s'
  end.
```

This predicate ensures R-ness of all the terms in the telescope once they are substituted by previous terms in the telescope. It needs to be in `Set` as R is in `Set`.

```
Fixpoint substitution_R Γ (s : substitution Γ) { struct s } : Set :=
  match s with
    | SNil ⇒ unit
    | SCons Γ' s' T t ⇒ (R (mult_subst [] t s') × substitution_R s')%type
  end.
```

Inversion of non-empty substitutions.

```
Lemma substitution_cons_aux : ∀ Γ (s : substitution Γ), ∀ Γ' T, Γ = Γ' , T →
  { (s', t) : substitution Γ' × term Γ' T | JMeq s (SCons s' t) }.

Lemma substitution_cons : ∀ Γ T (s : substitution (Γ , T)),
  { (s', t) : substitution Γ × term Γ T | JMeq s (SCons s' t) }.
```

Substitution has no effect on constants.

```
Lemma subst_cst : ∀ Γ c (s : substitution Γ),
```

mult_subst [] (cst Γ *c*) *s* = cst [] *c*.

Unfolding of telescope substitution on lambdas.

```
Lemma mult_subst_lam : ∀ Γ T T' (u : term (Γ , T) T') (s : substitution Γ),
  mult_subst [] (lam u) s = lam (mult_subst ([] , T) u s).
```

Unfolding of telescope substitution on applications.

```
Lemma mult_subst_app : ∀ Γ T T' (f : term Γ (T → T'))
  (e : term Γ T) (s : substitution Γ),
  mult_subst [] (app f e) s = app (mult_subst [] f s) (mult_subst [] e s).
```

Commutation of multiple substitutions is needed to prove that R is substitutive. It is just an application of the substitution lemma proved previously.

```
Lemma mult_subst_comm : ∀ Γ (s : substitution Γ) Δ T' T
  (u : term (Γ ; Δ , T') T) (s0 : term (Γ ; Δ) T'),
  subst (mult_subst (Δ , T') u s) (mult_subst Δ s0 s) =
    mult_subst Δ (subst u s0) s.
```

We extend *subst_lift* to contexts using substitution of telescopes and lifting by contexts.

```
Program Definition mult_subst_lift_stmt :=
  ∀ Δ (s : substitution Δ) T (t : term [] T),
    (mult_subst [] (lift_ctx Δ t) s) = t.
```

We need to have a dynamic coercion to prove these lemmas.

```
Program Fixpoint coerce_subst_context Γ (s : substitution Γ)
  Γ' (Heq:Γ = Γ') {struct s} : substitution Γ' :=
  match Γ' with
    | [] ⇒ SNil
    | Γ' , T' ⇒
      match s with
        | SNil ⇒ !
        | SCons _ s' T t ⇒
          @SCons Γ' (coerce_subst_context s' _) T' t
      end
  end.
```

Extension of the *on_coerce_proof* tactical.

```
Ltac on_coerce_proof tac :=
  match goal with
    [ | - ?T ] ⇒
    match T with
      | context [ coerce_var_context _?p ] ⇒ tac p
      | context [ coerce_term_context _?p ] ⇒ tac p
      | context [ eq_rect ____?p ] ⇒ tac p
```

```
      | context [ coerce_subst_context _?p ] ⇒ tac p
    end
  end.
```

The rest is just reinstantiation of tactics with the new *on_coerce_proof*.

We can finaly prove the mult_subst lemma.

```
Lemma mult_subst_lift : mult_subst_lift_stmt.
```

5.2 Main lemma.

A little lemma on variables is all that is missing before proving our main lemma. We can destruct variables, keeping the equality handy.

```
Lemma var_case_aux : ∀ Γ' T (v : var Γ' T), ∀ Γ t, Γ' = Γ , t →
  (t = T ∧ JMeq v (first Γ T)) + { v' : var Γ T | JMeq v (next v' t) }.

Lemma var_case : ∀ Γ t T (v : var (Γ , t) T),
  (t = T ∧ JMeq v (first Γ T)) + { v' : var Γ T | JMeq v (next v' t) }.
```

We now prove the main lemma, substitutivity of R. Note that we end with a closed term, escaping the problem of variables.

```
Lemma R_substitutive : ∀ Γ U (u : term Γ U) (s : substitution Γ),
  substitution_R s → R (mult_subst [] u s).
```

We get weak normalization as a trivial consequence of the previous lemma and *R_eval*.

```
Lemma term_wn : ∀ T (t : term [] T), { v : term [] T | evaluates t v }.
```

```
End Normalization.
End Λ.
```

6 Instantiation.

We will now instantiate Λ with the naturals and booleans. Our constants are naturals or booleans of *Coq*.

```
Inductive cst_types : Set := cst_nat_t | cst_bool_t.
Inductive csts : Set :=
  | cst_nat : nat → csts
  | cst_bool : bool → csts.

Definition type_constants (cst : csts) : cst_types :=
  match cst with
    | cst_nat _ ⇒ cst_nat_t
    | cst_bool _ ⇒ cst_bool_t
  end.
```

```
Module C.
  Definition constant_types : Set := cst_types.
  Definition constants : Set := csts.
  Definition type_constant : constants → constant_types := type_constants.
End C.

Module LambdaC := Λ C. Import LambdaC.
```

We construct trivial interpretations of the naturals and booleans in *Coq*.

```
Definition interp_constant_type (cst_t : cst_types) : Set :=
  match cst_t with
    | cst_nat_t ⇒ nat
    | cst_bool_t ⇒ bool
  end.

Definition interp_constant (c : csts) : interp_constant_type (type_constants c) :=
  match c return interp_constant_type (type_constants c) with
    | cst_nat n ⇒ n
    | cst_bool b ⇒ b
  end.
```

Wrap interpretation using the previous functions.

```
Definition interp_nb_type := interp_type interp_constant_type.
Definition interp_nb := interp_term interp_constant.
Definition interp_closed_nb := interp_closed_term interp_constant_type interp_constant.
```

6.1 An example term.

```
Definition nat_type : type := type_cst cst_nat_t.
```

The identity on naturals.

```
Program Definition nat_id_term : term [] (nat_type → nat_type) :=
  lam (rel (first [] nat_type)).

Eval compute in (interp_closed_nb nat_id_term).
Eval compute in (interp_type interp_constant_type (nat_type → nat_type)).
```

We can extract the term normalization function from this development.

```
Extraction "nbe.ml" term_wn.
```

References

Biernacka, Malgorzata, Danvy, Olivier, & Støvring, Kristian. (2006). Program extraction from proofs of weak head normalization. *Electr. notes theor. comput. sci.*, **155**, 169–189.